Arithmétique des Ordinateurs et Preuves Formelles

Guillaume Melquiond

Arithmétique des Ordinateurs et Preuves Formelles

Guillaume Melquiond

Inria, LRI, Univ. Paris-Saclay
guillaume.melquiond@inria.fr

Résumé

Cet article complète l'exposé du même nom et en reprend les grandes idées. L'exposé s'intéresse aux liens entre arithmétique des ordinateurs et vérification automatique, que ce soit pour de la preuve de programmes ou de théorèmes, le tout dans le cadre formel de l'assistant de preuve Coq. L'exposé est construit en deux parties.

La première s'intéresse à la preuve automatique de théorèmes mathématiques à l'aide de méthodes venues de l'arithmétique des ordinateurs. Il s'agit d'abord d'implanter et de formaliser une arithmétique à virgule flottante simplifiée mais efficace afin d'approcher les calculs réels. Puis une arithmétique d'intervalles peut être implantée au-dessus, offrant un moyen fiable de calculer des bornes sur des expressions à valeurs réelles. L'arithmétique d'intervalles dans sa version naïve est certes fiable mais rarement efficace à cause de l'effet de corrélation. L'étape suivante est donc de construire des approximations polynomiales fiables pour réduire cet effet. En combinant tout cela avec de la preuve par réflexion, il est alors possible de prouver automatiquement des bornes fines sur des expressions à valeurs réelles. Le procédé peut être poussé jusqu'à des intégrales propres voire même impropres. Tout cela a été intégré à la bibliothèque CoqInterval.

La deuxième partie s'intéresse à la vérification formelle d'algorithmes qui mettent en oeuvre de l'arithmétique à virgule flottante. C'était déjà le cas de certains des algorithmes de la première partie, mais l'utilisation de l'arithmétique d'intervalles les rendait en grande partie corrects par construction. Par contre, si l'on s'intéresse à des bibliothèques efficaces d'approximation de fonctions mathématiques (les « libm »), il n'y a plus rien de cela. Sans une vérification formelle, il est alors difficile de se convaincre que le code ne contient pas de nombreux bogues subtils dus à l'arithmétique à virgule flottante. Il est alors nécessaire d'effectuer une analyse fine des erreurs d'arrondi commises afin de s'assurer que les valeurs calculées par le code approchent correctement la fonction mathématique souhaitée. Mais une telle analyse est longue et difficile, surtout dans un cadre formel. C'est pour cela que l'outil Gappa a été conçu. Il permet de vérifier automatiquement des propriétés sur des algorithmes flottants, par une combinaison d'arithmétique d'intervalles, de réécriture et d'analyse de l'erreur directe. Il est aussi capable de générer des preuves formelles pour décharger des buts Coq.

1 Introduction

L'arithmétique à virgule flottante est une arithmétique bien adaptée au calcul en machine et son utilisation très majoritaire consiste en l'approximation d'opérations sur les nombres réels. Inspirée de la notation scientifique, elle offre une plage étendue de valeurs tout en garantissant un nombre conséquent de chiffres significatifs. Ainsi, le format *binary64* est capable de représenter des nombres de 10^{-308} à 10^{308} avec au moins 15 chiffres décimaux significatifs. La norme internationale IEEE 754 décrit précisément les formats et les opérations arithmétiques [8]. Cette norme étant très suivie, il est possible d'effectuer des calculs flottants dans un très grand nombre d'environnements.

On pourrait donc croire que, pour ce qui est d'effectuer des calculs numériques en machine, au moins d'un point de vue arithmétique, la question est réglée. Malheureusement, ce n'est pas

le cas. Tout d'abord, même si la plage de valeurs semble immense, nous ne sommes pas à l'abri d'un programme qui, lors de ses calculs, sortirait de cette plage, au moins temporairement. Ainsi, si la norme d'un vecteur excède 10^{154}, la façon la plus naturelle de calculer cette norme va provoquer un débordement de capacité. Dans ce cas, le calcul flottant va renvoyer $+\infty$, ce qui est assez loin de la valeur attendue.

Un problème plus pernicieux est la question de la qualité numérique des calculs (*accuracy* en anglais, à ne pas confondre avec *precision*). A priori, tout semblait parfait pourtant. Quinze chiffres décimaux sont en effet suffisants pour représenter précisément la plupart des valeurs intéressantes en pratique (oublions les problématiques monétaires). Qui plus est, la norme IEEE-754 garantit que, si le résultat d'une opération flottante n'est pas exactement représentable, le nombre représentable le plus proche du résultat exact doit être renvoyé. Ce nombre le plus proche sera représenté dans la suite par $\circ(\bullet)$. Malheureusement, ce n'est pas parce que chaque opération intermédiaire est correcte avec quinze chiffres que le résultat final l'est. Pour s'en convaincre, il suffit de considérer le calcul approché de $(2^{60}+1)-2^{60}$. La première somme $2^{60}+1$ est très bien approchée par $\circ(2^{60}+1) = 2^{60}$. Par contre, la somme finale $(2^{60}+1)-2^{60} = 1$ est infiniment mal approchée par $\circ(\circ(2^{60}+1)-2^{60}) = \circ(2^{60}-2^{60}) = 0$.

Il existe de nombreux exemples de calcul flottant ayant eu des conséquences désastreuses. Le plus connu, car le plus dramatique, causa la mort de 28 personnes et de nombreux blessés en 1991. Durant la Première Guerre du Golfe, l'armée des États-Unis installa des systèmes Patriot pour intercepter les missiles Scud visant ses bases. Mais la très grande vitesse des missiles Scud rendait l'interception hasardeuse et la décision fut prise d'augmenter la précision de certains calculs de trajectoire (mais pas tous). Cette modification provoqua une dérive des erreurs de calcul. Laisser tourner le système quelques jours sans le redémarrer fut suffisant pour que le missile intercepteur rate sa cible d'une fraction de seconde [12].

Ce qui rend cet exemple particulièrement intéressant est que le système s'est effondré parce que des personnes ont cherché à augmenter la qualité des calculs. Cela montre à quel point les calculs numériques défient l'intuition [11]. Se pose alors naturellement la question de leur fiabilité. Comment s'assurer qu'un programme ne va provoquer aucun comportement exceptionnel, par exemple un débordement ? Comment s'assurer que le résultat calculé est suffisamment proche du résultat attendu pour être utilisable sans risque ?

La section 2 donne quelques définitions et propriétés préliminaires sur l'arithmétique à virgule flottante et l'arithmétique d'intervalles. La section 3 s'intéresse ensuite à des algorithmes arithmétiques simples qui permettent de prouver formellement et automatiquement des propriétés par le calcul numérique. Finalement, la section 4 montre comment vérifier des algorithmes bien plus subtils tels qu'on peut les trouver dans les bibliothèques de fonctions mathématiques.

2 Préliminaires

2.1 Arithmétique à virgule flottante

Pour un format donné, les nombres flottants représentent des nombres réels de la forme $m \cdot \beta^e$. Les entiers m, β et e sont le signifiant, la base et l'exposant du nombre. La base est fixée, généralement 2 ou 10. Pour des raisons matérielles, les valeurs de m et e sont contraintes. En règle générale, nous assimilerons un nombre flottant au nombre réel qu'il représente.

Pour simplifier la formalisation, nous ne considérons que des formats $\mathcal{F}$ pour lesquels il existe une fonction $\varphi \in \mathbb{Z} \to \mathbb{Z}$ telle que

$$\mathcal{F} = \{x \in \mathbb{R} \mid x \cdot \beta^{-\varphi(\mathrm{mag}(x))} \in \mathbb{Z}\}$$

Format	β	ϱ	$e_{\min}$
binary32	2	24	-149
binary64	2	53	-1074
binary128	2	113	-16494
decimal32	10	7	-101
decimal64	10	16	-398
decimal128	10	34	-6176

TABLE 1 – Paramètres des formats décrits par la norme IEEE-754.

avec $\mathrm{mag}(x) = \lfloor \log_\beta |x| + 1 \rfloor$.

Deux familles de formats nous intéressent plus particulièrement ici. Les formats FLX sont décrits par des fonctions $\varphi(e) = e - e_{\min}$ tandis que les formats FLT sont décrits par $\varphi(e) = \max(e - \varrho, e_{\min})$. Cette dernière famille de formats est suffisante pour représenter les formats flottants classiques, si l'on fait abstraction des problèmes liés aux débordements de capacité. La table 1 indique comment choisir les paramètres ϱ et $e_{\min}$. Ceci étant dit, de nombreuses autres fonctions φ sont possibles, donnant des formats plus ou moins exotiques [4, §3.1.3].

La norme IEEE-754 indique que chaque opération flottante doit se comporter comme si elle calculait d'abord le résultat infiniment précis puis elle l'arrondissait au format de destination. Cela justifie l'introduction d'un opérateur d'arrondi $\square(\bullet)$. Une somme flottante entre deux nombres flottants u et v sera ainsi notée $\square(u + v)$.

Les opérateurs d'arrondi qui nous intéressent ont la forme suivante :

$$\square(x) = \lfloor x \cdot \beta^{-\varphi(\mathrm{mag}(x))} \rceil \cdot \beta^{\varphi(\mathrm{mag}(x))},$$

avec $\lfloor \bullet \rceil$ une fonction partie entière.

Si l'on choisit la partie entière inférieure, on obtient l'arrondi vers $-\infty$, tandis que si l'on choisit la partie entière supérieure, on obtient l'arrondi vers $+\infty$:

$$\begin{aligned} \bigtriangledown(x) &= \lfloor x \cdot \beta^{-\varphi(\mathrm{mag}(x))} \rfloor \cdot \beta^{\varphi(\mathrm{mag}(x))}, \\ \bigtriangleup(x) &= \lceil x \cdot \beta^{-\varphi(\mathrm{mag}(x))} \rceil \cdot \beta^{\varphi(\mathrm{mag}(x))}. \end{aligned}$$

Enfin, si l'on choisit l'entier le plus proche du réel (avec priorité à l'entier pair dans le cas ambigu), on obtient l'arrondi au plus près, au sens de la norme IEEE-754.

2.2 Arithmétique d'intervalles

Dénotons $\mathbb{I}$ les sous-ensembles fermés et connectés de $\mathbb{R}$. Il s'agit de $\emptyset$ et des intervalles $(-\infty; v]$, $[u; +\infty)$ et $[u; v]$ avec $u \leq v$ des nombres réels. Dans ce qui suit, nous nous intéresserons principalement aux intervalles dont les deux bornes sont des nombres flottants.

On dira d'une fonction $\mathbf{f} \in \mathbb{I}^n \to \mathbb{I}$ qu'elle est une extension par intervalles de $f \in \mathbb{R}^n \to \mathbb{R}$ si elle satisfait la propriété d'inclusion :

$$\forall \mathbf{x}_1 \in \mathbb{I}, \ldots, \mathbf{x}_n \in \mathbb{I},\ \forall x_1 \in \mathbb{R}, \ldots, x_n \in \mathbb{R},$$
$$x_1 \in \mathbf{x}_1 \wedge \ldots \wedge x_n \in \mathbf{x}_n \Rightarrow f(x_1, \ldots, x_n) \in \mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_n).$$

Une propriété intéressante des opérateurs d'arrondi vers $\pm\infty$ est $\bigtriangledown(x) \leq x \leq \bigtriangleup(x)$. Par conséquent, si l'on a deux encadrements $u \in [\underline{u}; \overline{u}]$ et $v \in [\underline{v}; \overline{v}]$, on peut en déduire les encadre-

ments suivants par monotonie des opérateurs arithmétiques :

$$\begin{aligned} u+v &\in [\nabla(\underline{u}+\underline{v}); \triangle(\overline{u}+\overline{v})], \\ u-v &\in [\nabla(\underline{u}-\overline{v}); \triangle(\overline{u}-\underline{v})], \\ u\cdot v &\in [\min(\nabla(\underline{u}\cdot\underline{v}), \nabla(\underline{u}\cdot\overline{v}), \nabla(\overline{u}\cdot\underline{v}), \nabla(\overline{u}\cdot\overline{v})); \\ &\quad\ \max(\triangle(\underline{u}\cdot\underline{v}), \triangle(\underline{u}\cdot\overline{v}), \triangle(\overline{u}\cdot\underline{v}), \triangle(\overline{u}\cdot\overline{v}))]. \end{aligned}$$

Cela permet donc de programmer très facilement des extensions par intervalles des opérateurs arithmétiques réels en quelques opérations flottantes. Et comme la propriété d'inclusion est préservée par composition, il est donc facile de borner une expression réelle arbitrairement compliquée.

Cependant, la propriété d'inclusion garantit seulement que l'encadrement $f(\vec{x}) \in \mathbf{f}(\vec{\mathbf{x}})$ est correct. Elle ne dit absolument rien de la finesse de l'intervalle $\mathbf{f}(\vec{\mathbf{x}})$. En particulier, celui-ci peut être trop large pour en déduire une propriété intéressante sur $f(\vec{x})$. Par exemple, de $x \in [0;1]$, on peut déduire $x - x \in [0;1] - [0;1] = [-1;1]$ par arithmétique d'intervalles. C'est un encadrement correct mais assez médiocre puisque $x - x = 0$. C'est le phénomène de perte de corrélation dû à la présence d'occurrences multiples de x.

3 Preuve par le calcul numérique

Comme indiqué dans la section précédente, l'arithmétique d'intervalles offre une façon simple de borner des expressions à valeurs réelles. Voyons voir comment transposer cela dans le cadre formel d'un système comme Coq. L'objectif est d'arriver à prouver formellement et automatiquement une propriété comme

$$\int_{-\infty}^{+\infty} \frac{(0.5 \cdot \log(\tau^2 + 2.25) + 4.1396 + \log \pi)^2}{0.25 + \tau^2}\, d\tau \leq 226.844. \tag{1}$$

3.1 Arithmétiques

La première étape consiste à formaliser une arithmétique à virgule flottante. La bibliothèque Flocq fournit une formalisation multi-base et multi-format de cette arithmétique [3][4, §3]. Mais l'important est qu'elle ne fournit pas seulement des opérateurs d'arrondi non calculables $\square(\bullet) \in \mathbb{R} \to \mathbb{R}$; elle fournit aussi des algorithmes formellement prouvés pour effectuer les opérations flottantes de base.

Dans le cas de la multiplication flottante de $m_1 \cdot \beta^{e_1}$ par $m_2 \cdot \beta^{e_2}$, l'algorithme est assez simple. Il commence par calculer le résultat exact $(m_1 m_2) \cdot \beta^{e_1+e_2}$. Puis il calcule un décalage d qui dépend de φ, de $e_1 + e_2$ et du nombre de chiffres de $m_1 m_2$ en base β. Et enfin il renvoie $\lfloor m_1 m_2 \beta^{-d} \rceil \cdot \beta^{e_1+e_2+d}$. Pour des opérations flottantes comme la division et la racine carrée, les algorithmes sont un peu plus subtils puisque le résultat exact n'est pas représentable sous la forme $m \cdot \beta^e$ [4, §3.3.2].

Une fois que l'on sait effectuer toutes les opérations flottantes de base, il est facile de construire une arithmétique d'intervalles effective sur le modèle de celle de la section 2.2. La bibliothèque CoqInterval fournit une telle arithmétique ainsi que tous les théorèmes d'inclusion correspondants [10]. Nous avons maintenant suffisamment de matériel pour prouver automatiquement et formellement au sein de Coq une propriété comme $u \cdot (v + \sqrt{v}) \geq -13$ avec $u \in [-3;4]$ et $v \in [1;2]$. Pour aller un peu plus loin, il nous faut enrichir le catalogue de fonctions disponibles.

Considérons le cas de la fonction arctan. Elle est monotone croissante, donc pour $x \in [\underline{x}; \overline{x}]$, nous avons $\arctan x \in [\arctan \underline{x}; \arctan \overline{x}]$. Pour calculer chacune des bornes de l'intervalle, nous pouvons utiliser la décomposition en série entière de arctan en 0:

$$\arctan x = x \cdot \sum_{i=0}^{\infty} (-1)^i \cdot \frac{(x^2)^i}{2i+1}.$$

Le premier problème est que le rayon de convergence de cette série ne permet de l'utiliser que si $|x| < 1$. Le second problème est que la série est infinie ; il faut donc la tronquer et pouvoir borner la valeur du reste. Fort heureusement, les termes tronqués étant décroissants en valeur absolue et de signes alternés, il suffit d'évaluer le premier terme tronqué pour borner leur somme infinie. Le dernier problème est que cette série converge lentement, donc se restreindre à $|x| < 1$ ne suffit pas, il vaut mieux avoir $|x| \leq \frac{1}{2}$ pour obtenir des bornes précises en un temps raisonnable. Pour cela, nous commençons par faire une réduction d'argument. La fonction étant impaire, nous nous ramenons au cas $x \geq 0$ puis nous utilisons les identités suivantes pour traiter le cas $x \geq \frac{1}{2}$:

$$\arctan x = \begin{cases} \frac{\pi}{4} + \arctan \frac{x-1}{x+1} & \text{pour } x \in [\frac{1}{2}; 2], \\ \frac{\pi}{2} - \arctan \frac{1}{x} & \text{pour } x \geq 2. \end{cases}$$

Pour obtenir des extensions par intervalles de exp et log, l'approche est similaire mais avec une réduction d'argument un peu plus compliquée. Pour cos et sin, cela devient encore plus compliqué puisque les fonctions ne sont pas monotones.

3.2 Approximations polynomiales et intégration

Maintenant que nous avons les briques de base pour borner de façon garantie l'intégrande de la formule (1), voyons comment passer à l'intégrale. Ignorons pour l'instant le fait que c'est une intégrale impropre et supposons plutôt que le but d'intégrer une fonction f entre $u \in \mathbf{u}$ et $v \in \mathbf{v}$. Quelques manipulations d'inégalités conduisent à l'encadrement suivant :

$$\int_u^v f \in (\mathbf{v} - \mathbf{u}) \cdot \mathbf{f}(\mathrm{hull}(\mathbf{u}, \mathbf{v})).$$

Malheureusement, cet encadrement est très grossier. Il donne par exemple $\int_{-1}^{1} x\,dx \in [-2; 2]$. Nous pouvons améliorer les choses en subdivisant l'intervalle d'intégration $[u; v]$ et en utilisant la relation de Chasles. Par exemple, $\int_{-1}^{1} x\,dx = \int_{-1}^{0} x\,dx + \int_{0}^{1} x\,dx \in [-1; 0] + [0; 1] = [-1; 1]$. Cela permet d'obtenir des bornes arbitrairement fines sur l'intégrale. Mais cette approche n'est en général pas utilisable à cause du nombre démesurément élevé de subdivisions nécessaires.

Le problème est ici que l'arithmétique d'intervalles approche le graphe d'une fonction par des rectangles ayant des côtés alignés avec les axes. Si la fonction n'est pas (presque) constante, le résultat est désastreux. Une meilleure approche consiste, étant donné $x_0 \in [u; v]$, à rechercher un polynôme p et un intervalle $\mathbf{\Delta}$ tels que

$$\forall x \in [u; v],\ f(x) - p(x - x_0) \in \mathbf{\Delta}.$$

Nous pouvons alors calculer P une primitive de p. Soit $\mathbf{P}$ une extension par intervalles de P. Nous obtenons ainsi

$$\int_u^v f \in \mathbf{P}(\mathbf{v}) - \mathbf{P}(\mathbf{u}) + (\mathbf{v} - \mathbf{u}) \cdot \mathbf{\Delta}.$$

Plutôt que de subdiviser $[u;v]$, nous pouvons augmenter le degré de p pour réduire la largeur de Δ et donc améliorer la qualité de l'encadrement final. La question est donc de savoir comment construire ces polynômes p. CoqInterval utilise pour cela des modèles de Taylor [10]. La terminologie vient du fait que, si f est une fonction de base comme exp, alors le polynôme correspondant est obtenu par la formule de Taylor-Lagrange :

$$\forall x \in [u;v],\ \exists \xi \in [u;v],\ f(x) = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x-x_0)^k + \frac{f^{(n+1)}(\xi)}{(n+1)!}(x-x_0)^{n+1}.$$

Et si f est une fonction composée, alors son polynôme s'obtient par composition de polynômes et un peu d'arithmétique d'intervalles donne $\mathbf{\Delta}$.

Il nous reste alors à régler le cas des intégrales impropres. Supposons cette fois que le but soit de borner $\int_u^{+\infty} fg$ avec g une fonction intégrable de signe constant sur $[u;+\infty)$. Encore une fois, quelques manipulations d'inégalités permettent d'obtenir un encadrement :

$$\int_u^{+\infty} fg \in \mathbf{f}(\mathrm{hull}(\mathbf{u}, +\infty)) \cdot \int_u^{+\infty} g.$$

Dans le cas de la formule (1), la fonction g à choisir est $\log^2 \tau/\tau^2$ dont CoqInterval sait borner l'intégrale impropre [9]. CoqInterval s'appuie sur la bibliothèque Coquelicot pour toutes les définitions et théorèmes d'intégration [2].

4 Preuve de calcul numérique

Dans la section 3, l'arithmétique à virgule flottante s'appuyait sur des formats de type FLX et les fonctions élémentaires étaient implantées de façon assez naïve. Tout cela est bien adapté pour faire de la preuve formelle par le calcul mais c'est finalement assez peu représentatif des algorithmes qui sont présents dans les bibliothèques de fonctions mathématiques. En particulier, quand le format flottant est fixé, de nombreuses optimisations sont possibles. La figure 1 présente ainsi une approximation flottante relativement efficacement de la fonction exp [6, p. 65].

La structure générale de la fonction est la même que celle présentée en section 3.1 : une réduction d'argument ramène l'entrée x dans un petit intervalle autour de zéro, puis la fonction y est évaluée, et enfin une reconstruction inverse l'effet de la réduction d'argument. Mais dans les détails, cela n'a pas grand chose à voir avec la façon dont CoqInterval implante exp. Premièrement, la réduction d'argument calcule un entier k et un nombre flottant $t \simeq x - k \cdot \log 2$ tel que $|t| \leq 355 \cdot 2^{-10}$. Deuxièmement, ce n'est pas la série entière tronquée de exp qui est évaluée mais une fonction rationnelle sortie du chapeau. Troisièmement, il n'y a aucun intervalle nulle part, donc ce n'est pas la propriété d'inclusion qui garantira la correction du résultat. Il va falloir prouver que la valeur calculée est proche de la valeur exacte, et cela en prenant en compte toutes les erreurs d'arrondi. Plus précisément, l'objectif est de prouver formellement que, si la toute dernière opération ne provoque ni *underflow* ni *overflow*, alors l'erreur relative finale est inférieure à 2^{-51}.

4.1 Erreur de méthode

Il est généralement possible de considérer séparément deux types d'erreur quand on analyse un algorithme flottant. D'un côté il y a l'erreur de méthode, c'est-à-dire l'erreur commise entre le résultat idéal et le résultat de l'algorithme si ce dernier était exécuté avec une précision

```
double cw_exp(double x) {
  if (x < -746.) return 0.;
  if (x > 710.) return INFINITY;
  // argument reduction
  double k = nearbyint(x * InvLog2);
  double t = x - k * Log2h - k * Log2l;
  // rational function evaluation
  double t2 = t * t;
  double p = 0.25 + t2 * (p1 + t2 * p2);
  double q = 0.5  + t2 * (q1 + t2 * q2);
  double f = t * (p / (q - t * p)) + 0.5;
  // reconstruction
  return ldexp(f, (int)k + 1);
}
```

FIGURE 1 – Approximation de exp au format *binary64*.

infinie. De l'autre côté il y a l'erreur d'arrondi commise entre des exécutions de l'algorithme avec des précisions soit infinie soit fixée.

Commençons par l'erreur de méthode. Elle concerne ici la fonction rationnelle qui a été choisie pour approcher exp. L'objectif est de borner la distance entre exp et $2f$ avec

$$f(t) = \frac{t \cdot p(t^2)}{q(t^2) - t \cdot p(t^2)} + 0.5,$$

où p et q sont des polynômes de degré 2 à coefficients *binary64*. Plus précisément, l'objectif est de prouver formellement

$$\forall t \in \mathbb{R},\ |t| \leq 355 \cdot 2^{-10} \Rightarrow \left| \frac{2f(t) - \exp t}{\exp t} \right| \leq 23 \cdot 2^{-62}.$$

Cette formule a une structure adaptée pour être prouvée par arithmétique d'intervalles. Malheureusement, comme $2f$ est choisie pour approcher précisément exp, c'est un cas typique de perte de corrélation (section 2.2). La Terre risque donc d'être engloutie par le Soleil avant même que la preuve soit terminée.

Fort heureusement, CoqInterval ne sait pas seulement calculer des encadrements à base d'intervalles, il sait aussi en calculer à base de polynômes, comme expliqué en section 3.1. En fait, c'est précisément pour vérifier ce genre de propriétés que les modèles de Taylor ont été ajoutés à CoqInterval. Si nous demandons à la tactique `interval` d'utiliser des polynômes de degré 9, elle produit automatiquement une preuve que Coq vérifie en une poignée de secondes.

4.2 Erreur d'arrondi

La fonction f approche $\frac{1}{2}\exp$ très précisément, mais ce n'est pas elle qui sera effectivement exécutée. C'est une version $\tilde{f}$ de f dans laquelle chaque opération arithmétique sera arrondie et donc la source d'une petite erreur. L'objectif est donc maintenant de vérifier une borne fine sur l'accumulation de toutes ces erreurs.

Tout comme pour l'erreur de méthode, la propriété semble prouvable par arithmétique d'intervalles, mais l'évaluation par intervalles de $\tilde{f}(t) - f(t)$ subit, là encore, une perte de corrélation. Nous allons cette fois utiliser l'outil Gappa pour borner l'expression [7]. Cet outil est basé sur l'arithmétique d'intervalles, ce qui lui permet de générer des preuves formelles vérifiables par Coq, mais il est aussi capable de transformer les expressions pour limiter la perte de corrélation.

Considérons le cas d'une multiplication entre deux sous-expressions u et v. Une fois exécutée en arithmétique à virgule, ces sous-expressions seront entachées d'une erreur ; notons les $\tilde{u}$ et $\tilde{v}$. Si l'on évalue directement $\tilde{u} \cdot \tilde{v} - u \cdot v$ par arithmétique d'intervalles, toute corrélation sera perdue. La réécriture suivante permet de grandement limiter cette perte :

$$\frac{\tilde{u} \cdot \tilde{v} - u \cdot v}{u \cdot v} = \frac{\tilde{u} - u}{u} + \frac{\tilde{v} - v}{v} + \frac{\tilde{u} - u}{u} \cdot \frac{\tilde{v} - v}{v}.$$

Autrement dit, plutôt que d'avoir à borner directement l'erreur relative entre $\tilde{u} \cdot \tilde{v}$ et $u \cdot v$, cette réécriture permet de considérer séparément les erreurs relatives entre $\tilde{u}$ et u et entre $\tilde{v}$ et v. Si tout se passe bien, d'autres réécritures permettront alors de limiter la perte de corrélation pour ces sous-problèmes.

Concrètement, Gappa dispose d'une base de données de quelques centaines de théorèmes et il l'utilise pour saturer la négation du but jusqu'à en déduire une contradiction. Ces théorèmes sont choisis pour reproduire les raisonnements que l'on pourrait faire à la main pour vérifier des propriétés sur des algorithmes flottants. Cette saturation a lieu hors de Coq et seule la succession de théorèmes qui mène à une contradiction est rejouée en Coq.

4.3 Réduction d'argument

Pour l'instant, nous n'avons considéré qu'un petit intervalle autour de zéro. Il reste à expliquer pourquoi la réduction d'argument est correcte et ne provoque pas une explosion de l'erreur finale. Or il s'avère que cette réduction est extrêmement subtile. C'est même le principal intérêt du code de Cody et Waite.

Pour s'en rendre compte, il suffit de remplacer `(x - k * Log2h - k * Log2l)` par `(x - k * (Log2h + Log2l))`. A priori ça ne devrait pas beaucoup changer le résultat, tout en accélérant le code. Et pourtant, après une telle modification, la valeur calculée par la fonction n'a plus grand chose à voir avec l'exponentielle, pour certaines valeurs de x.

La particularité de cette réduction d'argument est que la constante `Log2h` a été choisie de telle sorte que la multiplication par k est en fait une opération exacte. Mais encore faut-il réussir à prouver que ça rend la réduction d'argument correcte. Nous allons à nouveau utiliser Gappa mais il va cette fois falloir l'aider. Par exemple, le phénomène de perte de corrélation fait que Gappa est incapable de borner finement

$$x - \lfloor x \cdot \mathtt{InvLog2} \rceil \cdot \mathtt{Log2h}.$$

Une façon de résoudre le problème est de fournir l'égalité $x = x \cdot \mathtt{InvLog2} \cdot \mathtt{InvLog2}^{-1}$ à Gappa. L'outil se retrouve alors à devoir borner

$$(x \cdot \mathtt{InvLog2}) \cdot \mathtt{InvLog2}^{-1} - \lfloor x \cdot \mathtt{InvLog2} \rceil \cdot \mathtt{Log2h},$$

qui est une différence entre deux expressions ayant une structure similaire. Les méthodes décrites dans la section 4.2 peuvent alors s'appliquer. Un autre exemple est l'expression

$$((x - k \cdot \mathtt{Log2h}) - k \cdot \mathtt{Log2l}) - (x - k \cdot \log 2).$$

Là encore il y a perte de corrélation. Cette fois, il faut remplacer $\log 2$ par $\texttt{Log2h} + \delta$ et distribuer un peu les opérations pour que Gappa se retrouve à devoir borner

$$((x - k \cdot \texttt{Log2h}) - k \cdot \texttt{Log2l}) - ((x - k \cdot \texttt{Log2h}) - k \cdot \delta).$$

Il suffit alors d'indiquer à Gappa quelle est la différence entre `Log2l` et $\delta = \log 2 - \texttt{Log2h}$, différence que `interval` n'a aucune difficulté à borner. Au final, il aura suffi de fournir trois indications (les deux égalités et cette borne) pour que l'outil vérifie automatiquement toutes les propriétés nécessaires de la réduction d'argument.

Il ne reste alors plus qu'à utiliser Gappa pour combiner l'erreur de méthode, l'erreur d'arrondi de l'évaluation de la fonction rationnelle et l'erreur de la réduction d'argument et ainsi finir la vérification formelle de la fonction.

5 Conclusion

Nous avons vu comment prouver formellement et (presque) automatiquement des propriétés sur des expressions aussi bien à valeurs réelles qu'à valeurs flottantes. Et dans un cas comme dans l'autre, la preuve s'appuie sur l'exécution d'un algorithme à base d'arithmétique d'intervalles à bornes flottantes. Ces algorithmes ont été vérifiés en Coq et sont exécutés au sein de Coq, dans le style traditionnel de la preuve par réflexion [5]. Dans un cas, la finalité est la preuve de théorèmes mathématiques, et dans l'autre, il s'agit de la vérification d'algorithmes numériques.

De nombreuses pistes sont à explorer pour améliorer les bibliothèques et outils plus avant. Dans le cas de CoqInterval, une des limitations est l'absence d'automatisation concernant les fonctions définies implicitement, par exemple par équation différentielle. Il s'agirait de concevoir et de vérifier des algorithmes capables de construire des approximations polynomiales de telles fonctions. Pour ce qui est de Gappa, les résultats vérifiés automatiquement peuvent être puissants mais au prix d'indications parfois absconses de la part de l'utilisateur pour limiter la perte de corrélation. L'utilisation de formes affines symboliques pour représenter les erreurs d'arrondi ou d'un mécanisme équivalent permettrait peut-être de diminuer le travail de l'utilisateur [13].

Il y a un point qui n'a pas été abordé. C'est bien beau de vérifier formellement une fonction C, mais qu'est-ce qui nous garantit que c'est effectivement elle qui sera exécutée au bout du compte ? Par exemple, un compilateur ne risquerait-il pas de faire l'optimisation décrite en section 4.3 ? C'est pour se prémunir de ce genre de problèmes que nous avons défini une sémantique précise pour le C et que nous avons prouvé formellement que le compilateur CompCert la préservait lors de la compilation [1].

Références

[1] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. Verified compilation of floating-point computations. *Journal of Automated Reasoning*, 54(2):135–163, 2015.

[2] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A user-friendly library of real analysis for Coq. *Mathematics in Computer Science*, 9(1):41–62, 2015.

[3] Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In Elisardo Antelo, David Hough, and Paolo Ienne, editors, *20th IEEE Symposium on Computer Arithmetic (Arith)*, pages 243–252, Tübingen, Germany, 2011.

[4] Sylvie Boldo and Guillaume Melquiond. *Computer Arithmetic and Formal Proofs*. ISTE Press – Elsevier, 2017.

[5] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In Martín Abadi and Takayasu Ito, editors, *3rd International Symposium on Theoretical Aspects of Computer Software (TACS)*, volume 1281 of *Lecture Notes in Computer Science*, pages 515–529, Sendai, Japan, 1997.

[6] William J. Cody, Jr. and William Waite. *Software Manual for the Elementary Functions*. Prentice-Hall, Englewood Cliffs, NJ, 1980.

[7] Marc Daumas and Guillaume Melquiond. Certification of bounds on expressions involving rounded operators. *ACM Transactions on Mathematical Software*, 37(1):1–20, 2010.

[8] IEEE Computer Society. IEEE standard for floating-point arithmetic. Technical Report 754-2008, 2008.

[9] Assia Mahboubi, Guillaume Melquiond, and Thomas Sibut-Pinote. Formally verified approximations of definite integrals. *Journal of Automated Reasoning*, 2018.

[10] Érik Martin-Dorel and Guillaume Melquiond. Proving tight bounds on univariate expressions with elementary functions in Coq. *Journal of Automated Reasoning*, 57(3):187–217, 2016.

[11] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Basel, 2nd edition, 2018.

[12] Robert Skeel. Roundoff error cripples Patriot missile. *SIAM News*, 25(4):11, 1992.

[13] Alexey Solovyev, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic Taylor expansions. In Nikolaj Bjørner and Frank de Boer, editors, *20th International Symposium on Formal Methods (FM)*, volume 9109 of *Lecture Notes in Programming and Software Engineering*, pages 532–555, Oslo, Norway, 2015.